Enabling Traceability in an MDE Approach to Improve Performance of GPU Applications

Antonio Wendell de Oliveira Rodrigues, Vincent Aranega, Anne Etien, Frédéric Guyomarc'H, Jean-Luc Dekeyser

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Enabling Traceability in MDE to Improve Performance of GPU Applications

Antonio Wendell de O. Rodrigues — Vincent Aranega —

Anne Etien — Frédéric Guyomarc'h — Jean-Luc Dekeyser

N° 7720

September 2011

Embedded and Real Time Systems

ISSN 0249-6399 ISRN INRIA/RR--7720--FR+ENG

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE
INRIA
centre de recherche LILLE - NORD EUROPE

Enabling Traceability in MDE to Improve Performance of GPU Applications

Antonio Wendell de O. Rodrigues [*] , Vincent Aranega [†] , Anne Etien [‡] , Frédéric Guyomarc'h [§] , Jean-Luc Dekeyser [¶]

Theme : Embedded and Real Time Systems
Équipes-Projets DaRT

Rapport de recherche n° 7720 — September 2011 — 31 pages

Abstract: Graphics Processor Units (GPUs) are known for offering high performance and power efficiency for processing algorithms that suit well to their massively parallel architecture. Unfortunately, as parallel programming for this kind of architecture requires a complex distribution of tasks and data, developers find it difficult to implement their applications effectively. Although approaches based on source-to-source and model-to-source transformations have intended to provide a low learning curve for parallel programming and take advantage of architecture features to create optimized applications, the programming remains difficult for neophytes. A Model Driven Engineering (MDE) approach for GPU intends to hide the low-level details of GPU programming by automatically generating the application from the high-level specifications. However, the application designer should take into account some adjustments in the source code to achieve better performance at runtime. Directly modifying the generated source code goes against the MDE philosophy. Moreover, the designer does not necessarily have the required knowledge to effectively modify the GPU generated code. This work aims at improving performance by returning to the high-level models, specific execution data from a profiling tool enhanced by smart advices from an analysis engine. In order to keep the link between execution and model, the process is based on a traceability mechanism. Once the model is automatically annotated, it can be re-factored by aiming performance on the re-generated code. Hence, this work allows us keeping coherence between model and code without forgetting to harness the power of GPUs. To illustrate and clarify key points of this approach, an experimental example taking place in a transformation chain from UML-MARTE models to OpenCL code is provided.

Key-words: Profiling,GPU,OpenCL,MDE,Traceability,MARTE

[*] wendell.rodrigues@inria.fr
[†] vincent.aranega@inria.fr
[‡] anne.etien@lifl.fr
[§] frederic.guyomarch@inria.fr
[¶] jean-luc.dekeyser@lifl.fr

Centre de recherche INRIA Lille – Nord Europe
Parc Scientifique de la Haute Borne
40, avenue Halley, 59650 Villeneuve d'Ascq
Téléphone : +33 3 59 57 78 00 — Télécopie : +33 3 59 57 78 50

Traçabilité dans MDE pour Améliorer la Performance des Applications GPU

Résumé : Graphics Processor Units (GPU) sont connus pour offrir de hautes performances et d'efficacité énergétique pour les algorithmes de traitement qui conviennent bien à leur architecture massivement paralléle. Malheureusement, comme la programmation paralléle pour ce type d'architecture exige une distribution complexe des tâches et des données, les développeurs ont des difficultés à mettre en oeuvre leurs applications de manière efficace. Bien que les approches basées sur les transformations source-to-source et model-to-source ont pour but de fournir une basse courbe d'apprentissage pour la programmation paralléle et tirer parti des fonctionnalités de l'architecture pour créer des applications optimisées, la programmation demeure difficile pour les néophytes. Une approche Model Driven Engineering (MDE) pour le GPU a l'intention de cacher les détails de bas niveau de la programmation GPU en générant automatiquement l'application à partir des spécifications de haut niveau. Cependant, le concepteur de l'application devrait tenir compte de certains ajustements dans le code source pour obtenir de meilleures performances à l'exécution. Modifiant directement le code source généré ne fait pas partie de la philosophie MDE. Par ailleurs, le concepteur n'a pas forcément les connaissances requises pour modifier efficacement le code généré par le GPU. Ce travail vise à améliorer la performance en revenant aux modèles de haut niveau, les données d'exécution spécifiques à partir d'un outil de profilage améliorée par des conseils intelligents d'un moteur d'analyse. Afin de maintenir le lien entre l'exécution et le modèle, le processus est basé sur un mécanisme de traçabilité. Une fois le modèle est automatiquement annoté, il peut être repris en visant la performance sur la réutilisation du code généré. Ainsi, ce travail nous permet de garder la cohérence entre le modèle et le code sans oublier d'exploiter la puissance des GPU. Afin d'illustrer et de clarifier les points clés de cette approche, nous fournissons un exemple se déroule dans une chaîne de transformation à partir de modéles UML-MARTE au code OpenCL.

Mots-clés : Profiling,GPU,OpenCL,MDE,Traçabilité,MARTE

1 Introduction

Advanced engineering and scientific communities have used parallel programming to solve their large-scale complex problems for a long time. Despite the high level knowledge of the developers belonging to these communities, they find hard to effectively implement their applications on parallel systems. Some intrinsic characteristics of parallel programming contribute to this difficulty, e.g. race conditions, memory access bottleneck, granularity decision, scheduling policy or thread safety. In order to facilitate programming parallel applications, developers have specified several interesting programming approaches.

To increase the application development, software researchers have been creating abstractions layers that help themselves to program in terms of their design intent rather than the underlying architectures, *e.g.*, CPU, memory, network devices. They shield themselves from the complexities of these architectures. As method to implement these abstraction layers, approaches based on Model Driven Engineering (MDE) have frequently been used as a solution to accelerate software development, in particular MDE compilers.

In general, MDE compilers have the same structure than traditional compilers, but at a different level. High-level models are taken as input for the MDE compiler and a specific source code language is produced as output. Dealing with high-level models gives to the model designer a twofold advantages: on the one hand it increases re-usability and on the other hand it hides specific low-level details of code generation from the designed model. In this context, the software is directly generated from the high-level models.

Obviously, the details hidden by the high-level modeling must be introduced by the MDE compiler to generate a usable software. These details depend highly on the used programming language and their complexity vary from a language to another. In 2008, the consortium managed by Khronos Group released the first specification to Open Computing Language (OpenCL) [11]. The OpenCL language is the first open, royalty-free, standard for general-purpose parallel programming of heterogeneous systems. It provides a uniform programming environment for software developers who want to write efficient and portable code for high-performance computing servers, desktop computer systems and handheld devices using a diverse mix of multi-core CPUs, GPUs, Cell-type architectures and embedded processors.

As MDE framework for parallel embedded systems, Gaspard2 [9] proposes, among others, an MDE compiler to several programming languages. The compiler takes UML models profiled with the MARTE standard profile [20] as input model and generates software for few target platforms to reach simulation and validation purposes. Among the different target languages, Gaspard2 includes an OpenCL branch introduced in [15]. Using such a framework, the designer can focus on the general software architecture without worrying about the OpenCL implementation details.

However, the generated software could produce low performance issues due to a poor model design, even if optimization stages are proposed or implemented by the MDE compiler. Indeed, automatic generation does not ensure application performances when the application is launched. In these cases, fine-tuning the generated code to improve the performances is a real need.

In an MDE approach, fixing the software implies applying modifications on one of the two artifacts: the designed models or the software source code.

Nevertheless, directly modifying the generated source code is a difficult task for the model designer that does not know details about the target platform. Actually, the designer does not necessarily have the knowledge to efficiently modify the code. Moreover, the designed models lose the synchronization with the source code. A good solution to keep the coherence between the designed models and the generated source code is to regenerate code from the model correctly modified. Thus, changes aiming to achieve better performance must be made directly in the designed input models.

However, two issues make this method difficult to initiate as solution. Firstly, when a performance issue is observed, it is hard to figure out which parts of the models are responsible for this issue [21]. Thus, we have to keep a link between the models, the performance observations and the runtime results. Secondly, even if problems in the models are found, efficiently modifying it is not an easy task. Indeed, the improvement must often take into account details of the target architecture, usually unknown and hidden to the designer, to provide better and more efficient changes in the models.

Among the different techniques proposed to assist the designer during the performance improvement phase, two categories of tools can be found. The first one deals with static estimation computed in the model, whereas the second one, called profiling, deals with dynamic information. Our contribution focuses on profiling category because of its ability to gather details from a real execution environment. Hence, the recovered information comes directly from the software execution rather than from an estimation computed from the input high-level models.

In this research report, we present an approach based on model traceability to automatically return details and performance measures obtained during the software execution directly in the designed models. The performance information is recovered from dedicated profiling tools returning important measures as the running time or memories access time. Having such information directly reported in the input models gives a first overview of performance issues to the model designer.

In addition to the profiling information, specific smart advices are proposed in order to assist, as well as possible, model designers to achieve better results. These smart advices are computed by using the measured performances and specification details of the execution hardware. They allow the designers to easily and quickly fine tune their models aiming performance improvements. Once the models modified according to the proposed approach, the software is then regenerated keeping coherence between the models and the code.

The research report is structured as follows: in section 2 we present the different formalisms and technologies used in the research report. In section 3, we discuss about the major works of the domain. In section 4 we present the MDE based compilation chain that is used in the research report as support for our approach integration. In section 5 we present our traceability mechanism which plays an important role in our approach before showing how the compilation chain, the traceability mechanism and the profiling information are fitted together in section 6. We then present our approach on an example in section 7 before concluding in section 8.

2 Background Review

2.1 GPU and OpenCL

OpenCL is a standard for parallel computing proposed by Apple and has been released by Khronos Group since the first specification. OpenCL consists of a language, API, libraries and a runtime system. As depicted in Figure 1, this standard is based on a platform model that divides a system into one host and one or several compute devices. The compute devices act as co-processors to the host. Compute devices are subdivided into multiple compute units (CUs), which are also subdivided into one or multiple processing elements (PEs). An OpenCL application is executed on the host (*e.g.*, CPU), which sends instructions, defined in special functions called *kernels*, to the device (*e.g.*, GPU). Thus, OpenCL does not mean programming only devices, but host and device.

In the context of programming, the OpenCL standard defines a data parallel and a task parallel programming model[1]. In the data parallel model, the device runs multiple instances of the kernel in parallel on distinct data. Each instance is called a work-item (WI). While all work-items run the same kernel, they may perform different instructions at a time and occasionally change the instruction path *i.e.*, *Single Program, Multiple Data* (SPMD). Work-items can be arranged in work-groups (WG). OpenCL defines indexing schemes by which a work-item can be uniquely identified through either a global ID, or a work-group ID together with a local ID. The work-groups are assigned to CUs, where the work-items of each group are run in parallel on the PEs. Normally, multiple work-groups are assigned to the same CU, and multiple work-items are assigned to a PE. Conceptually, both are executed in sequence, but an implementation can use the excess parallelism for hiding memory latency (by switching between work-groups or work-items, respectively). Synchronization of work-items is possible within a work-group only, and takes the form of a barrier. OpenCL has many similarities with NVIDIA's GPU programming model[1] CUDA [12], the most of the differences are in relation to terminology. For instance, in CUDA, work-items are called threads, and work-groups are called blocks.

OpenCL also defines a programming language for writing kernels. OpenCL is an extension of C. Kernels are executed within their own memory domain and may not directly access host main memory.

2.2 MDE and MARTE

Model-Driven Engineering (MDE) [14] aims to raise the level of abstraction in program specification and increases automation in program development. MDE addresses to use models at different levels of abstraction for developing systems, thus raising the level of abstraction in program specification. An increase of automation in program development is reached by using model transformations. Higher-level models are transformed into lower-level models until the model can be made executable using either code generation or model interpretation.

The UML profile for MARTE [20] extends the possibilities for modeling applications, architectures and their relations. Moreover, MARTE allows extending the performance analysis and task scheduling based on target platform

[1] In this context, model is not associated with MDE.

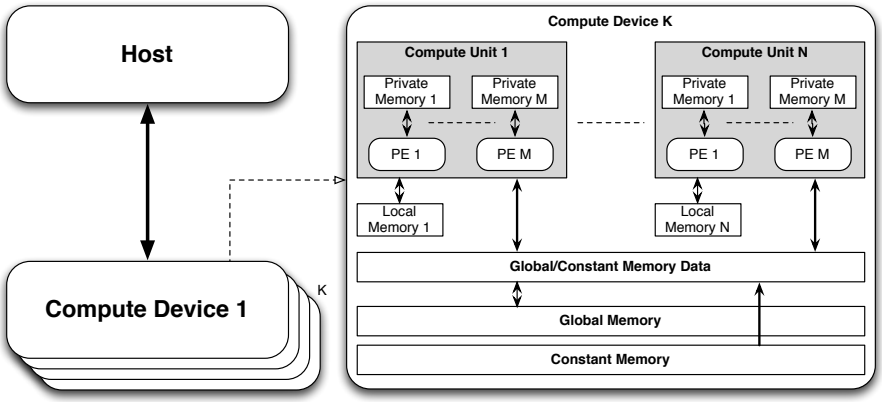


Figure 1: OpenCL Platform and Memory Model

architecture. MARTE consists in defining foundations for model-based description of Real-Time and Embedded Systems (RTES). As these core concepts have generic definition, they can be applied to GPUs as well. Among others, the benefits of using this profile are:

1. to provide a common way of modeling both hardware and software aspects of systems in order to improve communication between developers
2. to enable interoperability between development tools used for specification, design, verification, code generation
3. to foster the construction of models that may be used to make quantitative predictions regarding features of systems taking into account both hardware and software characteristics.

Allocation Modeling (Alloc) from *Foundations*, Generic Resource Modeling (GRM) and Generic Component Model (GCM) from *Design Model*, and Repetitive Structure Modeling are packages that provide the main resources to model and to describe our entire application. In particular, RSM provides concepts helping to express the inherent parallelism of applications.

3 Related Work

The analysis of software performance is part of the Software Performance Engineering (SPE) [22]. The SPE process uses multiple performance assessment tools depending on the state of the software and the amount of performance data available. SPE is a relatively mature approach and normally is associated to the prediction of the performance of software architectures during early design stages.

Several performance modeling approaches have been proposed in the literature, including simulation-based approaches and model-based approaches, particularly approaches based on UML. Some examples of works which have been developed in this research field are enumerated below.

1. Model-Driven SPE (MDSPE) is proposed in [23] and deals with building annotated UML models for performance, which can be used for the performance predictions of software systems. MDSPE consists in deriving the performance models from the UML specifications, annotated according to the OMG profile for Schedulability, Performance, and Time (SPT) [18]. It consists of *Performance Annotation* and *Performance Analysis* steps. The first one deals with encapsulation of performance characteristics of the hardware infrastructure, as well as QoS requirements of specific functions. The second one is implemented by a performance analyzer which computes the performance metrics, thereby predict the software performance.

2. A simulation-based software performance modeling approach for software architectures specified with UML is proposed in [2]. Similarly to MDSPE, this approach is also based on the OMG profile SPT [18], performance parameters are introduced in the specification model with an annotation. The performance results estimated by the execution of the simulation model are eventually inserted into the original UML diagrams as tagged values, so providing a feedback to the software designer. The proposed methodology has been implemented into a prototype tool called Software Architectures Performance Simulator.

3. The ArgoSPE approach, proposed in [10], is a tool for the performance evaluation of software systems in the first stages of the development process. From the designers viewpoint, ArgoSPE is driven by a set of "performance queries" that they can execute to get the quantitative analysis of the modeled system. For ArgoSPE, the performance query is a procedure whereby the UML model is analyzed to automatically obtain a predefined performance index. The steps carried out in this procedure are hidden to the designer. Each performance query is related to a UML diagram where it is interpreted, but it is computed in a petri network model automatically obtained by ArgoSPE.

4. An interesting approach aiming at model refactoring is depicted in [13]. This approach relies on optimization of parallel and sequential tasks on FPGA accelerators. In this case, before generating the VHDL code, the application at Register Transfer Level (RTL) is analyzed by an optimization system which exploits the allocation of FPGA resources. Then, the input model receives changes according to the optimization process results.

Further, there are several other research works that deal with SPE and UML but they are less co-related and we are not going into extended details. CB-SPE [3] reshapes UML performance profiles into component based principles. MDPE [7] describes a model transformation chain that integrates multi-paradigm decision support into multiple Process Modeling Tools. An extension [8] added to this work uses traceability and simulation engines in order to provide feedback to model designers. And in [21] is proposed a graph-grammar based method for transforming automatically a UML model annotated with performance information into a Layered Queueing Network (LQN) performance model.

As detailed in previous section, we base our work on the MARTE profile. It provides special concepts for performance analysis: Performance Analysis Modeling (PAM). This allows the model designer to define execution platform specification in the input models. However, as in UML SPT, the platform specification is modeled in the input model and assumes an infrastructure knowledge from the model designer. Moreover, the performance analysis is obtained from static estimations that may be far from performances measured at runtime.

In short, all these earlier works lack the profiling feedback and possible directions aiming better performances in a real execution environment. Besides, they do not take into account small differences in features of the target platform and their relation with application behavior. Our work does not require early annotations and relies on real execution environment aiming to make possible fine-tuning applications at design time.

4 From UML-MARTE to OpenCL

A new branch [15] of the Gaspard2 [9] framework was proposed to generate an effective OpenCL code. At design time, Gaspard2 uses UML profile for MARTE to refine the UML concepts of the application, then using transformation chains it allows the model designer to generate code for a few target platforms. One of the main advantages of MARTE is that it clearly distinguishes the hardware components from the software components. This is done via stereotypes provided in part by the Detailed Resource Modeling (DRM) package, in particular the *HwResource* stereotype. For hybrid (CPU and Compute Device) conception this separation is of prime importance as it is usual to create those two parts of the system simultaneously, by different teams. Moreover, this separation provides a flexible way to independently change the software part or the hardware part in a system co-design environment. For instance, this allows testing the software on different kinds of hardware architecture, or to reuse an architecture (with a few or no changes) for different applications.

The next subsections present the conceptual models defined at design time for the Gaspard2 framework.

4.1 Application Modeling

The conception of the *application* model relies on three main characteristics: first, what are the tasks and the interconnections among them; second, what is the repetition number of a task and its hierarchy that will be instantiated either in time or space[2]; and third, how to express the dataflow. Aiming at generating GPU code, to clearly distinguish a host from a compute device, both defined in OpenCL platform model, a tagged-value *description* in *HwResource* stereotype whose value is either *Host* or *Device*.

The allocation of the tasks onto processors is important during application design. Indeed, once we have an allocation model, model-to-model transformations can identify OpenCL kernels from input models' tasks which are mapped to the GPU ressources. The tasks allocation comprehends, both spatial distribution and temporal scheduling aspects, in order to map certain operations onto available computing and communication resources and services. For the allocation modeling purpose, the Allocation Modeling package of the MARTE profile is used.

Although MARTE is suitable for modeling purposes, it lacks the means to express the operations that a task will implement. Gaspard2 bridges this gap by introducing additional concepts and semantics to fill this requirement for system co-design. Gaspard2 defines a notion of a *Deployment* specification level [9] in order to generate compilable code from an application model. This level is related to the specification of a elementary component (EC): basic block having atomic functions. The deployment model allows the model designer to describe how the IPs (Intellectual Property), optimized and normally parametrized functions that depend of target technology, should be associated to ECs.

Under the perspective of data distribution, MARTE provides to the models special stereotypes based on Array-OL called *tilers* [4]. These stereotypes allows us specifying which data are processed by each iteration of our repetitive task.

[2]This relies on scheduling for clustered processors that involves spatial concerns (where to schedule) as well as temporal concerns (when to schedule).

4.2 Transformations

In MDE, a model transformation chain could be associated to a compilation process which transforms source models into target models through multiple transformations [24]. The source and the target models respectively conform to the source and the target metamodels. A model transformation chain relies on a transformation sequence where the output models of a transformation are used as input models for the next transformation. Such a decomposition makes easier the extension and the maintainability of a compilation process: new transformations extend the compilation process and each transformation can be modified independently from the others.

Figure 2 illustrates the OpenCL transformation chain of the Gaspard2 framework defined according to our model transformation engine. Each transformation is represented by a gear. For the UML-MARTE to OpenCL, the transformation chain is made of 9 transformation: 8 *model-to-model* transformations (gears numbered 1 to 8 in Figure 2) and one final *model-to-text* transformation (gear number 9 in Figure 2). The transformation engine runs transformations conform to the Query/View/Transformation standard (MOF QVT) [19], proposed by the OMG and uses the EMF framework [6] to manage the models. We have chosen QVTO as QVT implementation for the *model-to-model* transformation in Gaspard2.

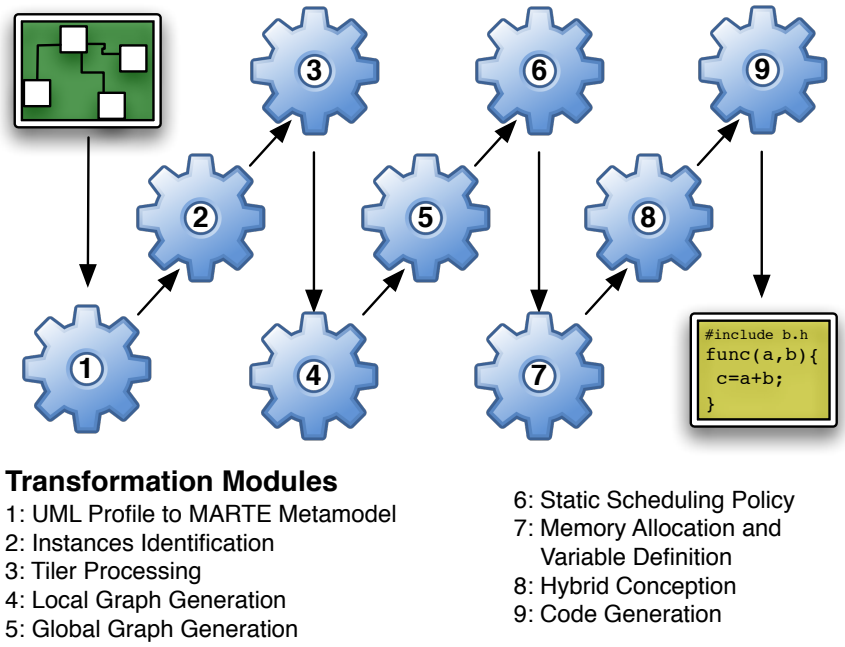


Figure 2: OpenCL Transformation Chain [15]

The input model in the Figure 2 represents the whole model (application, architecture, deployment, and allocation). Then, we no longer use UML profile

for MARTE. To make simpler transformations and to add further concepts, we use the MARTE metamodel whose elements come from stereotypes in the MARTE profile. Additional notions such as memory mapping are introduced by the transformation chain.

The other transformations take into account important aspects such as scheduling analysis. These transformations are more detailed in [15].

After the transformation execution number 8, a model with concepts closer to the target platform is produced. We call it hybrid model [15] because it contains all the artifacts useful to the generation of code on hybrid (or heterogeneous) architecture consisting of CPU and GPU. From this model, the code generation (*model-to-text* transformation number 9 in Figure 2) is a trivial step using Acceleo templates.

5 Traceability

In an MDE production chain, the model-to-model transformation execution uses elements from the input models to create elements in the output models. During the transformation execution, links between the elements of the input and output models can be gathered thanks to a traceability mechanism.

As we present in section 4, many transformations can be chained to produce a transformation chain. A trace model is produced for each model-to-model transformation. Thus, for a chain made of n transformations (including the final model-to-text transformation), $n-1$ model-to-model transformation traces are produced. In the case of UML to OpenCL transformation chain, 8 traces are produced. Using them, it is possible to recover what are the elements created in the $(n-1)$-th model from an element in the first model and *vice versa*. In this paper, we consider "ancestors" and "descendants". The ancestors of a model element X are the elements of the previous models in the transformation chain which lead to the creation of X. In the same way, the descendants of a model element Y are the elements of the next models in a transformation chain which are created from Y.

In the following subsections, we present our traceability mechanism which is based on two metamodels, the first one, named local trace, is used to keep the model-to-model transformation trace, whereas the second one, named global trace, is used to keep the sequence between the traces.

5.1 Local Trace

The local trace metamodel (presented in Figure 3) is designed around three main concepts: *Link*, *ElementRef* and *RuleRef*. A link binds two sets of elements: the *srcElements* and the *destElements*. The former set of elements leads to the creation of the latter set of elements. Each link may refer to a rule through the *RuleRef* concept when the trace is associated to a transformation. The other concepts of the metamodel are used to organize the trace models by providing containers. More details can be found in [1].

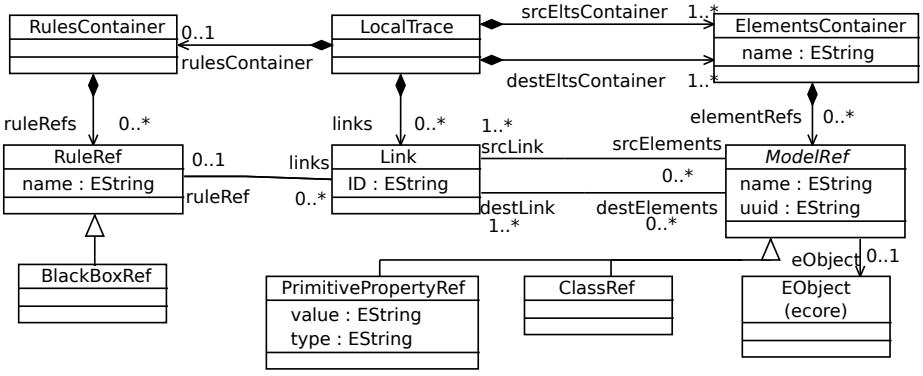


Figure 3: Local Trace Metamodel

5.2 Global Trace

If the local trace refers to a single transformation, the global trace eases the navigation between the different models and local traces in a transformation chain. Figure 4 shows the global trace metamodel.

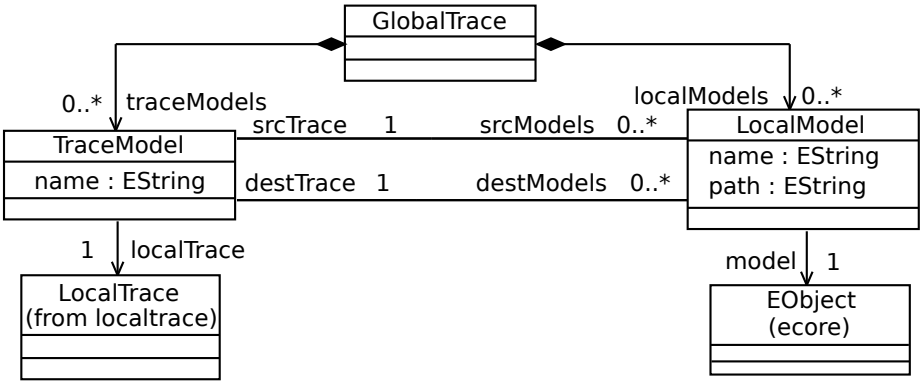


Figure 4: Global Trace Metamodel

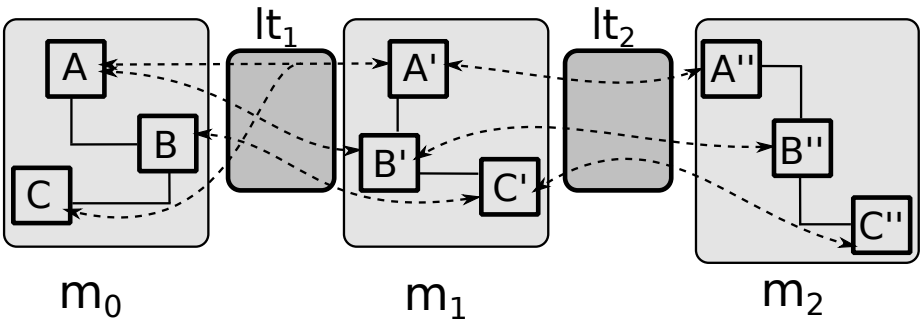
The global trace gathers references towards local trace and models used in a transformations chain while keeping the order induced by the chain. The local traces are represented by the *TraceModel* concept which is directly associated by the *localTrace* reference to a local trace root (the *LocalTrace* element from the local trace metamodel). Each *TraceModel* is linked to the models *LocalModel* used by the transformation.

5.3 The Need of a Reduced Trace

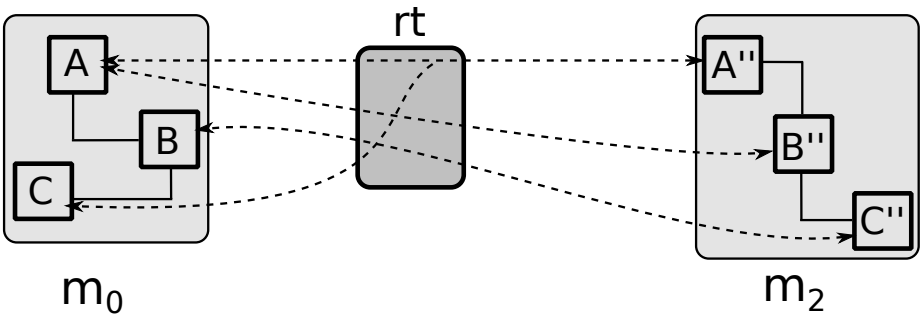
In performances analysis contexts, the intermediary traces and models generated are useless. The only required links are those between elements from the input models and the elements from the last one. The trace reduction allows the programmer to directly retrieve these links, without superfluous navigation. This reduced trace enables one to keep a reasonable and manageable trace size in a transformation chain.

The reduced trace is built by gathering, for each element of the last model, the elements of the input model leading to its creation. More precisely, the global trace model and the various local trace models are navigated in the backward direction. The reduced trace establishes a bridge between elements from the input model and elements of the last model exactly as if the transformation chain would be assimilated to a single transformation. Figure 5 shows the reduced trace principles within an example.

The model m_0 is transformed into a model m_1 and the trace lt_1 is produced. The model m_1 is then directly consumed by a transformation that translated it in the model m_2 and produced the trace lt_2. The two traces link the source and destination elements involved in the transformations. For example, the trace lt_1 expresses that the element A in the initial model m_0 enables to the creation of the element B' in the intermediary model m_1. This trace also shows that the A' element in the intermediate model m_1 has been created from the A and C elements. Then, the A' element leads to the creation of the A'' element in



(a) Unreduced Traces



(b) Reduced Trace

Figure 5: Trace Reduction Principle

the final model, whereas the *B'* element leads to the creation of the element *B"* (according to the trace lt_2).

The trace reduction algorithm searches for ancestors elements in the initial model of a given element in the last model. For the *A"* element, it navigates from *A"* in m_1 to *A'* in m_1. From *A'*, it navigates to *A* and *C* in m_0. Thus, the kept link puts *A* and *C* from m_0 in a direct relation with *A"*. The same algorithm is performed for each element in the last model. The resulting reduced trace is labeled rt in Figure 5(b).

6 Integration Approach

The main goal in this work is to provide a high level profiling environment in a model design context. Performance execution feedbacks are directly provided in the input models. From this way, the model designers can easily understand the designed system behavior and identify what should be fine tuned. The main interest of this approach is to benefit from real data from a profiling tool instead of using performance statically computed from performance constraints designed in the input models.

Our approach is sketched in Figure 6. The profiling life-cycle presented follows a classic structure usually manually performed:

1. the software is generated from the high-level models (step 1 and *cf.* section 4) and the trace models are produced
2. the software is executed, producing profiling logs (steps 2 to 4)
3. the produced logs are analyzed and returned in the input models (steps 5 to 7).

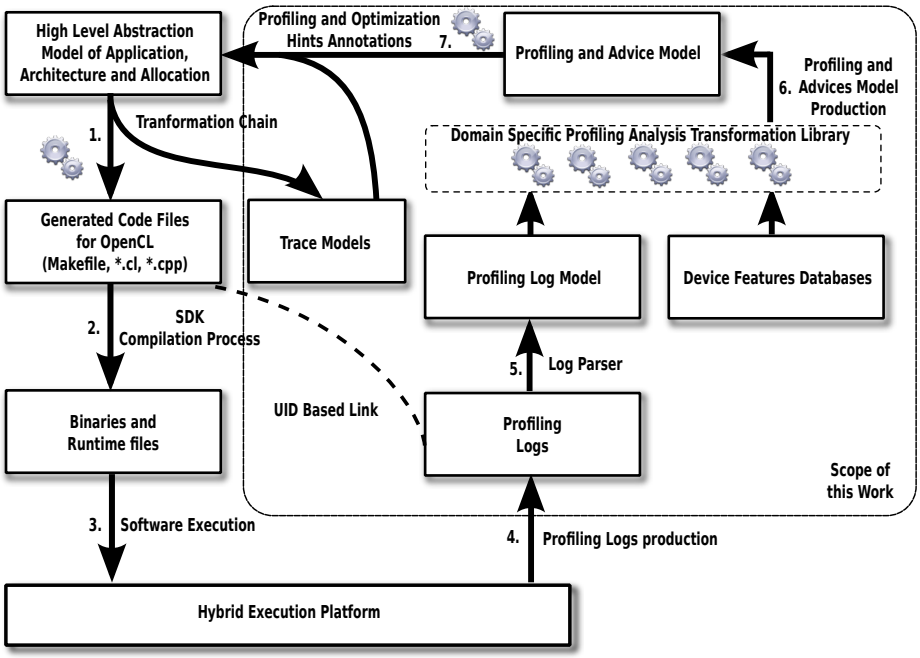


Figure 6: Approach Overview

Currently, in works available through the literature, only the first part of the process is automatic (step 1). The second part (steps 2 to 4) producing the logs highly depends on the used tools. The third part where logs are analyzed and connected to the input elements that should be modified remains a manual and complex process. In this paper, we focus on this step of the process (steps 5 to 7 in Figure 6) by automating it.

In order to automate this analysis and the performances feedback, our process uses two main artifacts: an expert system (reported as *Domain Specific Profiling Analysis Transformation Library* in Figure 6) and the model-to-model traceability (*Trace Model*). The expert system is used during the analysis step, whereas the traceability is used for the performance feedback.

In a first time, we present how the traceability is managed in the compilation chain and the required modifications on the model compilation chain. Then, we present the expert system creating the profiling advices and, finally, we show how these advices are reported into input models.

6.1 Managing The Whole Chain Traceability and Avoiding Model-to-Text Traceability

In order to keep the links between the input models and the software execution, trace models are produced all along the compilation chain, except for the model-to-text transformation. The translation from model to text implies keeping information on text blocks and words [17]. The granularity for this kind of trace made its management and maintainability difficult. In our case, the code has to be studied only in term of the abstract concepts from the models, and not in terms of blocks and words. Thus, in this paper, the model-to-text traceability has been avoided.

To bypass the model-to-text trace, the code generation deals with unique identifiers (UIDs) associated to each elements in the last model. The profiling logs produced by the software execution refers to the UID of the analyzed element. Thus, the *Profiling Logs* can be rebound to the model world.

Concretely, in order to generate the UIDs, we use the EMF feature called *Universal Unique IDentifier* (UUID) and, consequently, we modify the compilation chain. A new transformation adding the UID was inserted as last step of the model-to-model transformation chain, just before the code generation.

6.2 From Execution to Smart Advices

Once the software code is generated, the software is executed. During the execution, profiling logs are produced by third party tools.

6.2.1 Profiling Logs Parsing

According to the used Software Development Kit (SDK) and profiling tools, these profiling logs are generated with a dedicated format. This format is parsed using a shell-script that builds a profiling model that conforms to the metamodel presented in Figure 7.

The metamodel root: *ProfilingModel* gathers the different entries that can be found in the profiling logs. Each profiling entry from the logs is represented by a *LogEntry* gathering the hardware model (*e.g.* Tesla T10 or G80) with the *archiModel* attribute. Each *LogEntry* contains several *Parameter* elements owning a *kind* and a *data* representing: the information type (*e.g.*, occupancy, time or memory consumption), and its value. In order to keep the link between the profiling information and the transformation chain, each *LogEntry* keeps the UID from the logs with the *UID* attribute. In addition, a *timeStamp* attribute is added to the *LogEntry* in order to keep the logs sequence. This metamodel

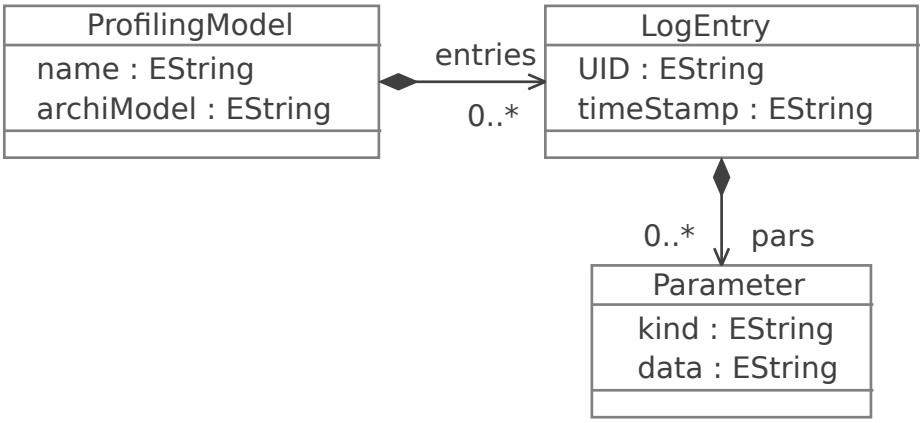


Figure 7: Profiling metamodel

is generic enough to produce models that can gather information that can be found in the profiling logs.

6.2.2 The Expert System

In-depth knowledge of the target platform is really important to identify which elements should be modified. Model designers do not always have such a knowledge. Thus, more than profiling results, we propose to provide smart advices to model designers. To reach this aim, we integrate an expert system that uses input data from two sources: *profiling logs* and *device features database*. The first one gives us factual data about execution. The second one gives us behavior features of the target platform. By combining both sources, it is possible to deduce what to do to attain some optimization level. For instance, assuming the device supports 32MB in shared memory allocation per group of work-items and the application allocates at runtime 48MB. The expert system is able to indicate that it is necessary to decrease the memory allocation after analyzing profiling log results and device constraints. In this case, the expert system provides a hint where the problem occurs. In order to analyze the many properties of the results, an extensible library (*cf.* Figure 6) is proposed in this paper.

The device features database gathers a set of devices from a specific vendor. It is represented as a model that conforms to the metamodel of Figure 8 in order to be properly handled by either transformation languages or other tools based on the EMF framework. In the context of the UML-MARTE to OpenCL transformation chain, the target platform is the GPU.

The model root is represented by *DeviceFeratures*. It gathers the many vendors' GPU models (*e.g.*, Tesla T10) represented by the *GPU_Models* concept. These GPU models are associated to a group of GPU devices (represented by the *GPU_Device* concept) having the same allocation granularity (*AG* attribute) and compute capability (*CC* attribute). Their values are specified according to two enumerations: *AllocationGranularity* and *ComputeCapability*, respectively. In the first enumeration, each literal represents the compute capability version of the GPU (*e.g.*, the literal *cc10* represents the 1.0 compute capability ver-

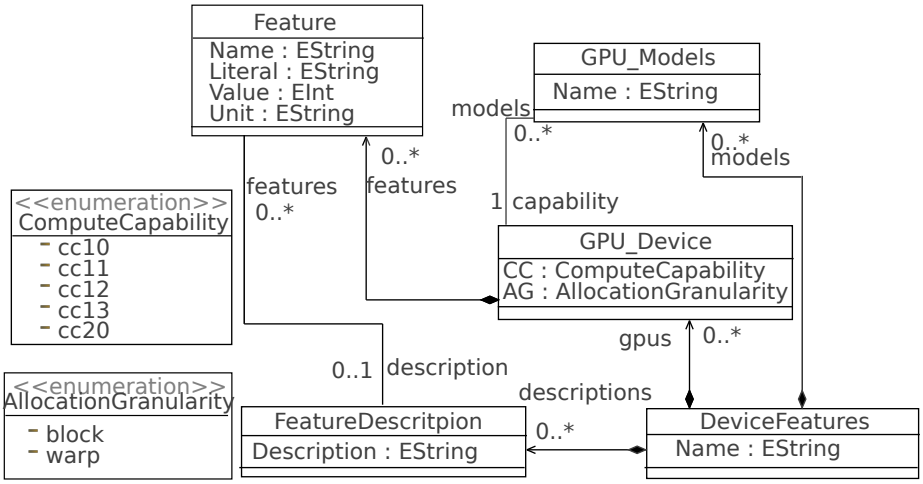


Figure 8: GPU Device Features metamodel

sion). In the second enumeration two choices are possible: the data allocation is performed either by *block*(work-groups) or by *warp*[3].

The different devices presented in the model contain various features and their descriptions (represented by *Feature* and *FeatureDescription*). So, for instance, the GPU Tesla T10 which has compute capability 1.3 has the feature *Work-Items per Warp* equals 32. Figure 11 in section 7 shows a model based on this metamodel.

6.3 Backtracking Advices in the Input Models

The advice as well as the computed value obtained during the software execution must be reported in the input model. Thus, the model designer can directly access the advice reported on the element requiring the modification. For the information feedback, the reduced trace produced during the transformation chain execution is used.

The UID contained in the computed advice refers to elements in the last model before the code generation. This UID comes from the *Profiling Logs* and it is retained even after transformations. Once the element referred by the UID is found, the reduced trace is backward navigated in order to recover the input elements producing the profiling information. Two cases can occur. The retrieved elements are reduced to one or several. Indeed, a simple element in the last model can be produced from either one or many elements in the input models.

Reporting the advice on all retrieved elements can create confusion. To solve this issue, the expert system can be configured to specify some element types of the input metamodel for each libraries. From this way, only the input elements of the specified types are kept. Finally, the computed advice is connected to these elements in the input models.

[3]A warp is a GPU related concept that indicates the number of work-items in a work-group ready to hardware scheduling. It is also known as *wavefront*.

In the UML-MARTE to OpenCL transformation chain, we decide to use the *Comment* concept from UML. Indeed, this concept element has the ability to gather an information in a string format. Moreover, *Comment* can be linked to any kind of UML element what places it as a perfect candidate for carrying the advice computed by the expert system. As the advice representation is quite dependent from the input metamodel and the input metamodel capabilities, it could be be provided using an other concept.

7 Example and Benchmarks

In order to illustrate the practical application of the approach proposed in this paper, we describe a case study. This case study is a complete example which presents an application design, code generation and the profiling feedback.

In this example the goal is to offer information and compute an advice about how improving the processors' occupancy. This will help the application designers to identify input elements parameters which they can modify aiming better results.

7.1 Vector Product Application

The example presented in this section is a vector product. For pedagogical reasons, we chose this simple operation that does not require further knowledge in more complex applications such as signal processing or numerical analysis to be understood. Nevertheless, it gathers all the relevant concepts to illustrate our approach. Moreover, we have also tested our approach on large scale examples like the classic downscaler algorithm [16]. The vector product is an algebraic operation that takes two equal-length(N) sequences of numbers and return another sequence obtained by multiplying corresponding entries. A sequential code sample in C language for this operation is showed in the listing 1.

Listing 1: Code Snippet for the Application

```
for (uint i=0; i<N; i++)
 c[i] = a[i] * b[i];
```

We have created the UML-MARTE model (see Figure 9) for this application using the Papyrus [5] modeling tool. Elementary tasks *TE_ genarray1*, *TE_ genarray2*, and *TE_ printarray* are responsible for generating and printing vectors. As these tasks comprehend the application interface (data input and output), they are run by the CPU.

The application's vectors are arrays of 16,000,000 elements. We have chosen this large number to take advantage of the massively parallel processors provided by GPUs. The composed component *ForDevices* instantiated in the program consists of a repetitive task *ep:TE_ elemprod*. In our application, this kind of task is composed by operations on single elements. Repetitive tasks are potentially parallel and are allocated onto GPU. The repetition shape of the task in this case is *{16,1000000}*, *i.e.* the task operation runs 16 millions times on one element of each vector whose size equals 16 millions. This shape has two dimensions: 16 and 1,000,000. The total of work-items is calculated by multiplying these two dimensions. The first one becomes the number of work-items and the second one the number of groups. The definition of this shape is a decision of the designers and usually they take into account the *Intellectual Property (IP)* interface associated to the elementary task and its external *tilers* (see subsection 4.1). Moreover, considering that, in compute capability 1.x devices, memory transfers and instruction dispatch occur at the Half-Warp(16 work-items) granularity, it is reasonable to define groups composed by 16 work-items at a first try.

Figure 10 presents the allocation process for the repetitive task, *i.e.* it defines which devices will manage a task. For instance, the *ep: TE_ elemprod*

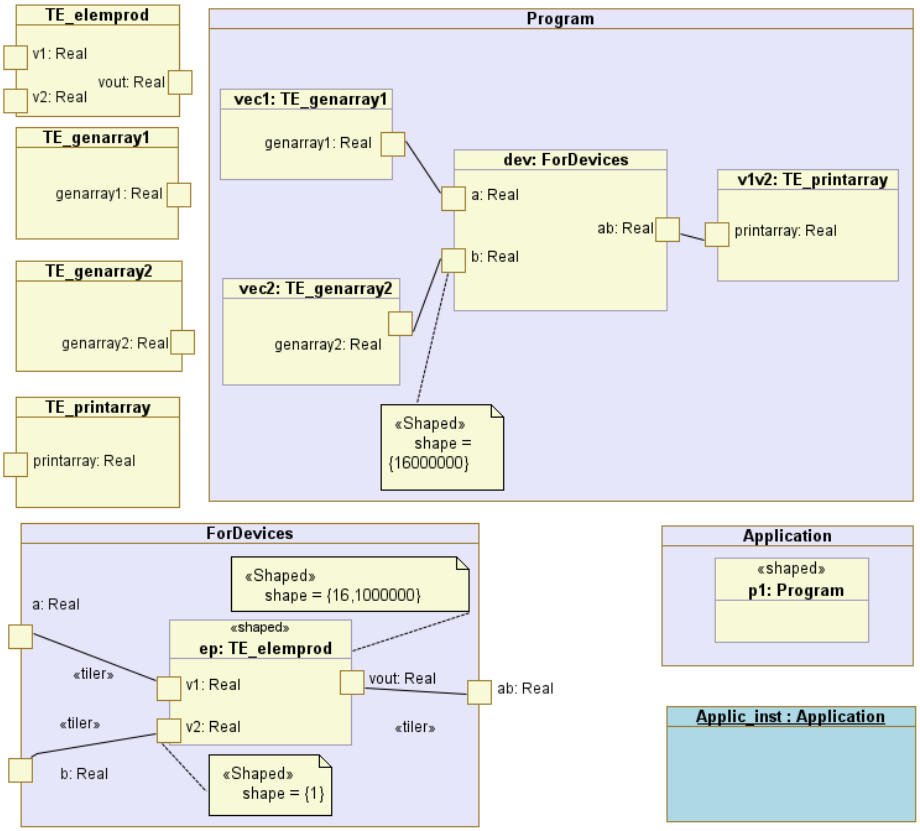


Figure 9: Vector Product Application Model

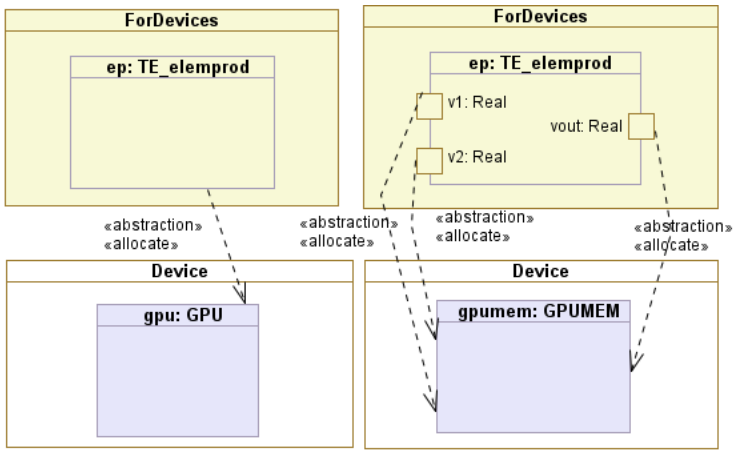


Figure 10: Task and Memory Allocations onto GPU

task (also visible in Figure 9) will be run by the GPU (*gpu: GPU* instance).

Similarly, the memory mapping process is also defined for the communication ports. According to Figure 10, the communication ports will be managed by the GPU memory (*gpumem: GPUMEM*). Thanks to the task allocation process, the model compiler identifies *OpenCL kernels*. The memory allocation step is also important because it creates *host* and *device* variables and organizes the data transfers.

Once the application is designed with all necessary well configured elements, we generate all source code files necessary to the target compiler. In addition, trace models are generated for each model-to-model transformation thanks to the traceability mechanism.

Listing 2 shows the code only for the *kernel*. This is a generated code composed of two functions: the *IP* function, represented on lines 1-4, and the *kernel* function (lines 6 to 60). The UID value (see section 6) is identified from the last generated model in the UML-MARTE to OpenCL transformation chain and is concatenated to the *kernel* name. From line 6 we identify, concatenated to *kernel* name, the UID ***_uCQs6obGEeCiXMyak_whYg***. This UID is the link between the code and the model elements whenever they have to be referenced.

Listing 2: Generated Kernel

```
1  void elemprod(const float* a, const float* b, float* c)
2  {
3    c[0]=a[0]*b[0];
4  }
5
6  __kernel void ep_KRN__uCQs6obGEeCiXMyak_whYg(
7  uint iNumElements,
8  const __global float* v2_ep_KRNPAR,
9  __global float* vout_ep_KRNPAR,
10 const __global float* v1_ep_KRNPAR)
11 {
12   float v1_ep[1]; float v2_ep[1]; float vout_ep[1];
13   //get index into globaldata array
14   int iGID = get_global_id(0) +
15   get_global_size(0)*get_global_id(1) +
16   get_global_size(0)*get_global_size(1)*get_global_id(2);
17   if (iGID < iNumElements) // bound check
18   {
19     { //input tiler
20       uint tlIter[2];
21       uint tl[1];
22       uint ref[1];
23       uint index[1];
24       tlIter[0]=iGID%16;
25       tlIter[1]=abs(iGID/16);
26       ref[0] = 0 + 1*tlIter[0] + 1*tlIter[1]*tlIter[0];
27       for(tl[0]=0; tl[0] < 1; tl[0]++) {
28         index[0]= (ref[0]+ 0*tl[0])%16000000;
29         v2_ep[tl[0] * 1] = v2_ep_KRNPAR[index[0] * 1];
30       }
31     }
32     { //input tiler
33       uint tlIter[2];
34       uint tl[1];
35       uint ref[1];
36       uint index[1];
37       tlIter[0]=iGID%16;
38       tlIter[1]=abs(iGID/16);
39       ref[0] = 0 + 1*tlIter[0] + 1*tlIter[1]*tlIter[0];
40       for(tl[0]=0; tl[0] < 1; tl[0]++) {
41         index[0]= (ref[0]+ 0*tl[0])%16000000;
42         v1_ep[tl[0] * 1] = v1_ep_KRNPAR[index[0] * 1];
43       }
44     }
45     elemprod(v1_ep, v2_ep, vout_ep); //IP call
46     { //output tiler uint
47       tlIter[2];
48       uint tl[1];
49       uint ref[1];
50       uint index[1];
51       tlIter[0]=iGID%16;
52       tlIter[1]=abs(iGID/16);
53       ref[0] = 0+ 1*tlIter[0] + 1*tlIter[1]*tlIter[0];
54       for(tl[0]=0; tl[0] < 1; tl[0]++) {
55         index[0]= (ref[0]+ 1*tl[0])%16000000;
56         vout_ep_KRNPAR[index[0] * 1]=vout_ep[tl[0] * 1];
57       }
58     }
59   }
60 }
```

The rest of the code consists of private variable declarations (line 12), a limit control to avoid overlapping data bounds (lines 14-17), two input *tilers* to gather the elements from global memory (lines 18-43), the *IP* call (line 45); and an output *tiler* writing the result into global memory (lines 47 to 56).

7.2 Profiling Feedback

We used the following configuration as platform for our tests:

- CPU AMD Opteron 8-core @2.4GHz and 64GB RAM;
- GPU NVidia S1070 4 devices Tesla T10 (4GB RAM each) - Compute Capability 1.3;
- Linux, GCC 4.1.2, OpenCL 1.0.

Among all the measures coming from the profiler, the kernel occupancy factor has an important impact on performance. Usually the aim at executing a kernel is to keep the multiprocessors and, consequently, the device as busy as possible. The work-items instructions are executed sequentially in OpenCL, and, as a result, executing other warps when one warp is paused or stalled is the only way to hide latencies and keep the hardware busy. Some metric related to the number of active warps on a multiprocessor is therefore important in determining how effectively the hardware is kept busy. This metric is called occupancy. The occupancy is the ratio of the number of active warps per multiprocessor (WPM) to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardwareâĂŹs ability to process warps that are actively in use. Hence, higher occupancy does not always equate to higher performance, there is a point above where additional occupancy does not improve performance. However, low occupancy always interferes with the ability to hide memory latency, resulting in performance degradation.

The important features to compute the occupancy and that vary on the different compute capability are:

- the number of registers available;
- the maximum number of simultaneous work-items resident on each multiprocessor;
- and the register allocation granularity.

The number of work-items resident on a multiprocessor relies on index space as known as *N-Dimensional Range (NDRange)*. The MARTE to OpenCL chain obtains the information from the shape of the task which will become a *kernel*. Hence, changes in the dimensions of shape affect the occupancy. From the point of view of the proposed approach, *occupancy* is a specialized module that can be included to the expert system. For other analysis other specialized module can be added to attain specific goals. For this example, we analyze the *occupancy* of the multiprocessors. Occupancy is function of constant parameters (features) from device and some measures directly obtained from the profiler.

The process of calculating occupancy is implemented in a QVT transformation. This transformation takes two input models (according to Figure 6): the *Device Features Database* and the *Profiling Logs*. In this example the first one

conforms to a metamodel based on NVidia GPUs (*cf.* Figure 8). Although this metamodel was designed according to vendor's features, it can be modified or extended to comply with other vendor device models. For instance, from the model presented in Figure 11 we see that the GPU Tesla T10 has compute capability equals 1.3 and is its warps contain 32 threads (or work-items in OpenCL terminology).

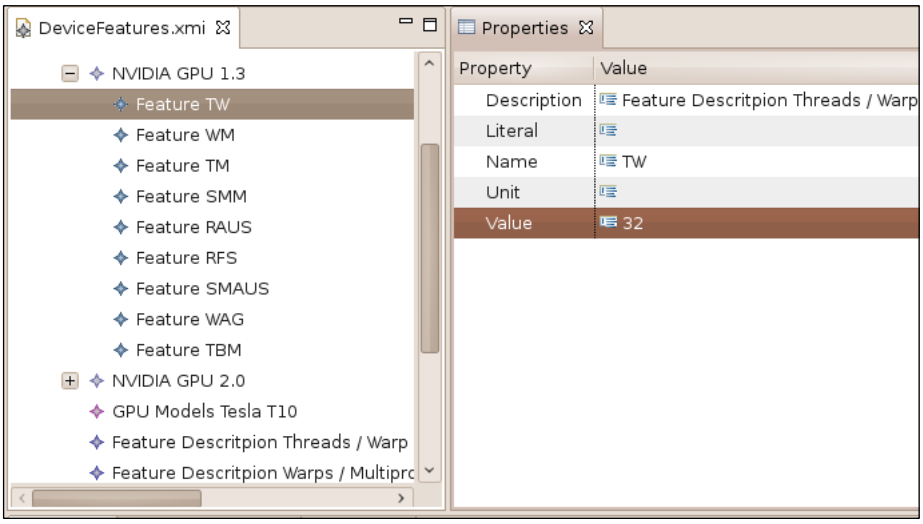


Figure 11: GPU Device Features Database Model

7.3 Benchmark

The profiling environment creates a log file in CSV format having some dynamic measured data (as seen in Figure 12). The file header contains data about the target platform. We deal with a Tesla T10 GPU in this case. The rest of the file consists in description of fields and log entries. The description of fields indicates in which order they will appear in a log entry. For instance, in Figure 12, a log entry begins with the *timestamp* field. The second field that can be retrieved in an entry is *gpustarttimestamp*, then *method* and the 13-th field that can be found in the log is the *occupancy* field. For our example, log entries about memory copies are not important (log entry with the *method* field sets to *memcpyHtoDasync*). The following entries in the log correspond to *kernel* calls. A shell-script parser takes this text file as input and converts it to XMI format that conforms to the profiling metamodel depicted in Figure 7. The model (Figure 13) created from the CSV log file gathers exactly th same information. Except the *timestamp* field and the UID contained in the *method* field that becomes attributes of the *LogEntry*, all the other fields (*e.g. gputime*, *cputime* or *ndrangesizex*) are transformed into *Parameter* with the value of the log entry. Figure 12 (highlighted elements) and Figure 13 present the occupancy parameter with value 0.250 for the kernel call with timestamp equals 1283955.000.

```
# OPENCL_PROFILE_LOG_VERSION 2.0
# OPENCL_DEVICE 0 Tesla T10 Processor
# OPENCL_PROFILE_CSV 1
# TIMESTAMPFACTOR fffff6f3dd57cd20
timestamp,gpustarttimestamp,method,gputime,cputime,ndrangesizeX,ndranges
izeY,workgroupsizeX,workgroupsizeY,workgroupsizeZ,stapmemperworkgroup,re
gperworkitem,occupancy,streamid,local_load,local_store,gld_request,gst_r
equest,memtransfersize,memtransferdir,memtransferhostmemtype
698239.000,1220f847c305ce40,memcpyHtoDasync,40347.039,41886.000,,,,,,,,
1,,,,,64000000,1,0
1241546.000,1220f847e367f960,memcpyHtoDasync,40539.457,41379.00
0,,,,,,,,,1,,,,,64000000,1,0
1283955.000,1220f847e5e6a2c0,ep_KRN__uCQs6obGEeCiXMyak_whYg,1238.752,147
0.000,65535,1,16,1,1,32,11,0.250,1,0,0,4370,2185
1285577.000,1220f847e5fee540,ep_KRN__uCQs6obGEeCiXMyak_whYg,1237.120,142
8.000,65535,1,16,1,1,32,11,0.250,1,0,0,4368,2184
1287138.000,1220f847e616bd40,ep_KRN__uCQs6obGEeCiXMyak_whYg,1237.600,143
7.000,65535,1,16,1,1,32,11,0.250,1,0,0,4370,2185
...
```

Figure 12: Sample profiling results in CSV format

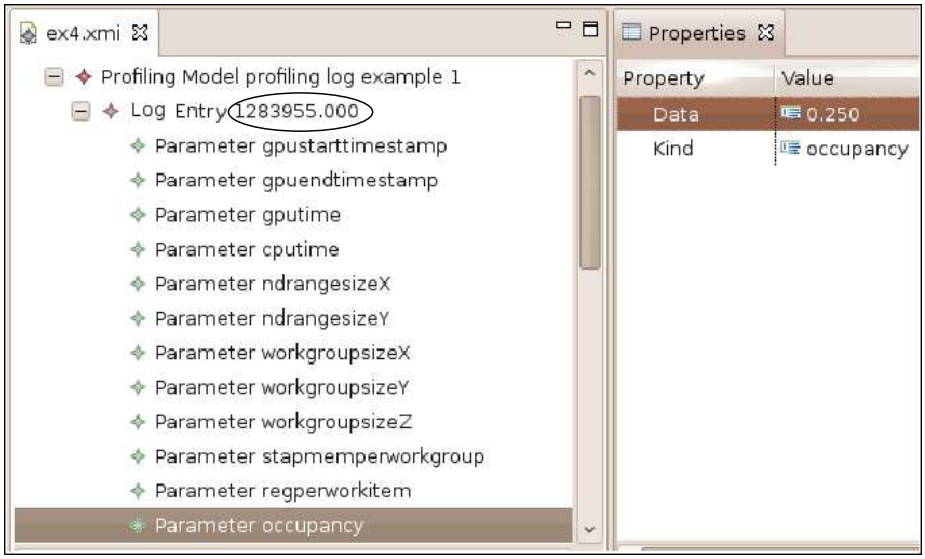


Figure 13: Profiling Results Model

Although our methodology does not impose a rigid workflow, our approach relies on two major activities: first we run the code exactly as it is generated from the original input model, then the application designer analyzes the runtime behavior based on profiling feedback annotated on the input model; second, the designer, taking into account the provided information and hints, modifies the model aiming to obtain better results. Once the model is modified, the code is again generated and then executed in order to verify the result of theses changes.

For this example the first running gives the results seen in Table 1. The application launches 1 million groups of 16 work-items onto the device. However,

a hardware limit imposes a maximum of 65535 groups by kernel. Thus, for the whole execution, 15 kernels are launched with the maximum of 65535 groups and 1 kernel is launched with 16975 groups. Summarizing, the last line in table shows that this configuration gives 16 kernels calls and only 25% in the multiprocessor occupancy. Moreover, this configuration take 2.55% of the GPU time. The other part of the time comprehends data transfers and idle states. The launch grid (NDRange) has a two-dimensional size (*i.e.*, how many groups and how they are organized). Each group has a three-dimensional size (represented by *[16 1 1]* on third column). However, only the first dimension is used in this example. Our goal is to increase the occupancy and decrease the relative GPU time.

Calls #	NDRange	WGSize	Occup.	GPUTime
15	[65535 1]	[16 1 1]	25%	
1	[16975 1]	[16 1 1]	25%	
16	[1000000 1]	[16 1 1]	25%	2.55%

Table 1: Profiling results for the non-optimized code

By using our approach results are combined with GPU features and this returns a smart advice as comment in the input UML-MARTE model (Figure 14). Besides the performance parameters available directly on the comment, a *hint* points out a possible change in the model to improve the generated code. Additionally, the advice provides an image reference of a chart (as seen in Figure 15) for all predicted occupancy according to these results. In this case it is suggested to change the task shape from $\{16, 1000000\}$ to $\{128, 125000\}$. A simple analysis seeks the first block size giving 100% on occupancy as seen in Figure 15. For instance, the expert system automatically highlights the first (block size=128) and second (block size=256) maximum values in the chart.

Calls #	NDRange	WGSize	Occup.	GPUTime
1	[65535 1]	[128 1 1]	100%	
1	[59465 1]	[128 1 1]	100%	
2	[125000 1]	[128 1 1]	100%	1.07%

Table 2: Profiling results for the new code

Table 2 presents the profiling results for the code generated from the modified input model. For this case, we have two kernel calls, 100% on occupancy and a reduction to 1.07% on the GPU time. As expected, the modified model achieves better performance than the original one. Figure 16, obtained from a visual profiler provided within NVidia tools, shows us that, without modifications, the whole execution of the kernel is about 146% slower.

For each operation, the first bar is related to the time measured from the optimized models, whereas the second bar is related to the time measured from the initial models. The transfer times presented in Figure 16 correspond to the following operations:

- *memcpyDtoHasync* is called once in both executions. This transfers the result vector from device to host.

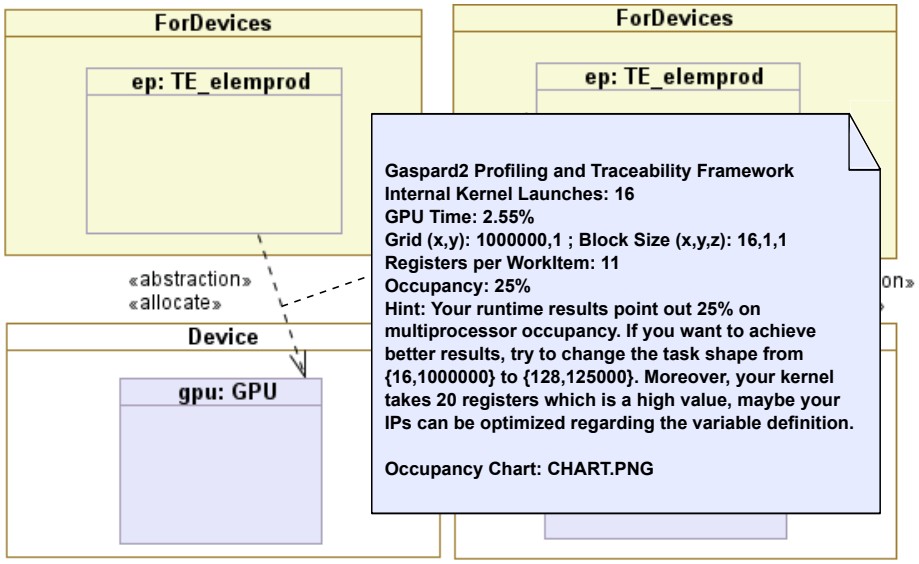


Figure 14: Annotated Model

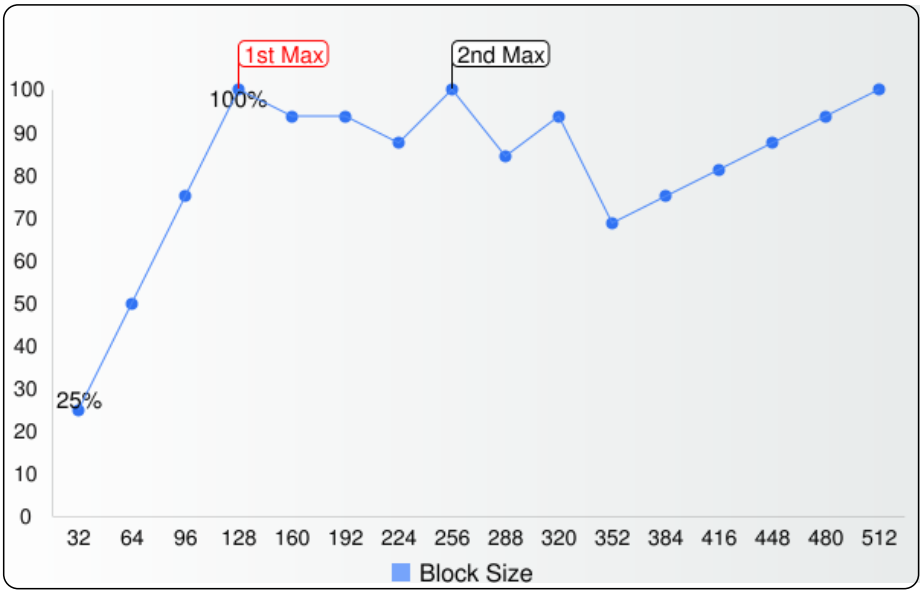


Figure 15: Occupancy by Varying Block Size

- *memcpyHtoDasync* is called twice in both executions. This transfers the two input vectors from host to device.

As we have changed only the task shape, transfer times do not have any expressive alteration.

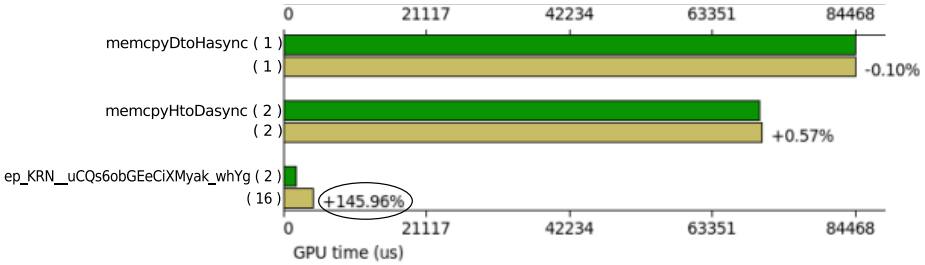


Figure 16: Comparison Summary Plot from Visual Profiler

8 Conclusion and Future Directions

In this paper we address performance improvements as part of the life cycle of the application design and execution. We provide a high level profiling environment for OpenCL in the context of the Gaspard2 framework. This environment allows the model designers to efficiently modify their models to achieve better performances. The profiling environment is based on two main artifacts: an expert system and a traceability mechanism.

The expert system proposed here uses data from a feature database dedicated to the runtime platform and profiling logs. The aim is to compute a smart advice explaining how the model should be modified in order to achieve better performances. From the point of view of the application designers, they do not necessarily need to know complex details about the runtime platform. Moreover, no performance specification is given in advance. The expert system summarizes the profiling logs and minimizes the tasks of the model designer by analyzing the gathered profiling data.

Specially for GPU applications, better performances rely on speed-up, memory use and processor occupancy. In order to provide this feedback, this paper proposed to retain coherence between code generation and traceability. Thus, the expert system uses the traceability to return a UML *Comment* consisting of a computed smart advice and profiling logs on the specifics input models elements that involve the analyzed performance issue.

Although our case study is focused on applications GPU, our approach can be adapted to other environments with code generation and profiling tools. Indeed, the library that is part of the expert system can be extended aiming to analyze other issues or other devices. Moreover, the traceability mechanism that we provide can fit to any transformation chain. Thus, the library of the expert system is the only artifact that requires additions in order to deal with other languages or other performance issues. For instance, memory bottleneck footprints in applications can be explicitly annotated onto links between ports. This helps developers to identify critical points in their applications.

As future direction for our approach, we aim a semi-automatic modification of the input models using the advices returned in the input model. Hence, input models could be incrementally modified until the performances are considered acceptable by the model designer.

References

[1] Vincent Aranega, Jean-Marie Mottu, Anne Etien, and Jean-Luc Dekeyser. Traceability Mechanism for Error Localization in Model Transformation. In *ICSOFT*, Bulgaria, July 2009.

[2] Simonetta Balsamo and Moreno Marzolla. A simulation-based approach to software performance modeling. *SIGSOFT Softw. Eng. Notes*, 28:363–366, September 2003.

[3] Antonia Bertolino and Raffaela Mirandola. Towards Component-Based Software Performance Engineering. In *Component-Based Software Engineering*, 2003.

[4] Pierre Boulet. Array-OL Revisited, Multidimensional Intensive Signal Processing Specification. Technical report, 2007.

[5] CEA. Papyrus - Open Source Tool for Graphical UML2 Modeling. `http://www.papyrusuml.org`.

[6] Eclipse Consortium. EMF. `http://www.eclipse.org/emf`, 2007.

[7] Mathias Fritzsche and Wasif Gilani. Model transformation chains and model management for end-to-end performance decision support. In *Proceedings of the 3rd international summer school conference on Generative and transformational techniques in software engineering III*, GTTSE'09, pages 345–363, Berlin, Heidelberg, 2011. Springer-Verlag.

[8] Mathias Fritzsche, Jendrik Johannes, Steen Zschaler, Anatoly Zherebtsov, and Alexander Terekhov. Application of Tracing Techniques in Model-Driven Performance Engineering. In *In Proccedings of the 4th ECMDA-Traceability Workshop (ECMDA'08*, June 2008.

[9] A. Gamatié, S. Le Beux, E. Piel, R. Ben Atitallah, A. Etien, P. Marquet, and J-L. Dekeyser. A Model Driven Design Framework for Massively Parallel Embedded Systems. *ACM Transactions on Embedded Computing Systems (TECS). (to appear)*, 2011.

[10] E. Gómez-Martínez and J. Merseguer. ArgoSPE: Model-based Software Performance Engineering. volume 4024, pages 401–410. Springer-Verlag, Springer-Verlag, 2006.

[11] Khronos Group. OpenCL - The open standard for parallel programming of heterogeneous systems. `http://www.khronos.org/opencl/`.

[12] David Kirk and Wen mei Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 1st edition, February 2010.

[13] Sébastien Le Beux. *Un Flot de Conception pour Applications de Traitement du Signal Systématique Implémentées sur FPGA à Base d'Ingénierie Dirigée par les Modèles*. These, Université des Sciences et Technologie de Lille - Lille I, December 2007.

[14] D. Lugato, J-M Bruel, and I. Ober. *Model-Driven Engineering for High Performance Computing Applications, Modeling Simulation and Optimization - Focus on Applications*. Shkelzen Cakaj (Ed.), 2010.

[15] A. Wendell O. Rodrigues, Frédéric Guyomarc'H, and Jean-Luc Dekeyser. An MDE Approach for Automatic Code Generation from MARTE to OpenCL. Technical report, INRIA Lille - RR-7525. http://hal.inria.fr/inria-00563411/PDF/RR-7525.pdf/.

[16] A. Wendell O. Rodrigues, Frédéric Guyomarc'H, and Jean-Luc Dekeyser. Programming Massively Parallel Architectures using MARTE: a Case Study. In *2nd Workshop on Model Based Engineering for Embedded Systems Design (M-BED 2011) on Date Conference 2011*, Grenoble, France, March 2011.

[17] Gøran Olsen and Jon Oldevik. Scenarios of Traceability in Model to Text Transformations. In *Model Driven Architecture- Foundations and Applications*, Lecture Notes in Computer Science. 2007.

[18] OMG. UML Profile for Schedulability, Performance, and Time, version 1.1. `http://www.omg.org/spec/SPTP`, 2005.

[19] OMG. M2M/Operational QVT Language. `http://www.eclipse.org/emf`, 2007.

[20] OMG. Modeling and Analysis of Real-time and Embedded systems (MARTE), Version 1.0. `http://www.omg.org/spec/MARTE/1.0/`, 2009.

[21] Dorina C. Petriu and Hui Shen. Applying the uml performance profile: Graph grammar-based derivation of lqn models from uml specifications. In *Proceedings of the 12th International Conference on Computer Performance Evaluation, Modelling Techniques and Tools*, TOOLS '02, pages 159–177, London, UK, 2002. Springer-Verlag.

[22] Connie U. Smith and Lloyd G. Williams. *Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software*. Addison Wesley Longman Publishing Co., Inc, 2002.

[23] I. Traore, I. Woungang, A.A. El Sayed Ahmed, and M.S. Obaidat. UML-based Performance Modeling of Distributed Software Systems. In *Performance Evaluation of Computer and Telecommunication Systems (SPECTS)*, pages 119 –126, july 2010.

[24] Bert Vanhoof and Yolande Berbers. Breaking up the transformation chain. In *Proceedings of the Best Practices for Model-Driven Software Development at OOPSLA 2005*, San Diego, California, USA, 2005.

Contents

Centre de recherche INRIA Lille – Nord Europe
Parc Scientifique de la Haute Borne - 40, avenue Halley - 59650 Villeneuve d'Ascq (France)

Centre de recherche INRIA Bordeaux – Sud Ouest : Domaine Universitaire - 351, cours de la Libération - 33405 Talence Cedex
Centre de recherche INRIA Grenoble – Rhône-Alpes : 655, avenue de l'Europe - 38334 Montbonnot Saint-Ismier
Centre de recherche INRIA Nancy – Grand Est : LORIA, Technopôle de Nancy-Brabois - Campus scientifique
615, rue du Jardin Botanique - BP 101 - 54602 Villers-lès-Nancy Cedex
Centre de recherche INRIA Paris – Rocquencourt : Domaine de Voluceau - Rocquencourt - BP 105 - 78153 Le Chesnay Cedex
Centre de recherche INRIA Rennes – Bretagne Atlantique : IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex
Centre de recherche INRIA Saclay – Île-de-France : Parc Orsay Université - ZAC des Vignes : 4, rue Jacques Monod - 91893 Orsay Cedex
Centre de recherche INRIA Sophia Antipolis – Méditerranée : 2004, route des Lucioles - BP 93 - 06902 Sophia Antipolis Cedex

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt, BP 105 - 78153 Le Chesnay Cedex (France)
http://www.inria.fr
ISSN 0249-6399